Practical assignment on Concurrent Programming 2023/2024

Nelma Moreira José Proença

Department of Computer Sciences, Faculty of Sciences, University of Porto

What to submit

A *PDF report* that answers the proposed questions **and** a zip file with the developed *implementations* in pseuCo (or caal) and in Scala. The implementations should include a `readme.txt` explaining how to run the implementation.

Groups

This is a group assignment. Groups should have at most 2 members – if you fail to find a partner please let us know.

How to submit

A link will be made available in moodle: `https://moodle2324.up.pt/course/view.php?id=1748`. Only one of the group members has to submit the assignment, but both should be well identified in the main report.

Presentation

Present your encodings and implementations in a 8 minute presentation, to be scheduled for **June 3** - **June 4**, 2024.

Deadline

23h59m of **May 29**, 2024 (Wednesday)

Mutual exclusion

Exercise 1. Recall below Hyman's algorithm (1966) from the first module, for the mutual exclusion between two processes. Assume initially $\texttt{k} = 1$.

$$P_1 = \left[\begin{array}{l} \texttt{while true do} \\ \quad \texttt{noncricital actions} \\ \quad \texttt{b}_1 \leftarrow \texttt{true}; \\ \quad \texttt{while k} \neq 1 \texttt{ do} \\ \quad\quad \texttt{while b}_2 \texttt{ do} \\ \quad\quad\quad \texttt{skip}; \\ \quad\quad \texttt{k} \leftarrow 1 \\ \quad \texttt{critical actions} \\ \quad \texttt{b}_1 \leftarrow \texttt{false} \end{array}\right] \qquad P_2 = \left[\begin{array}{l} \texttt{while true do} \\ \quad \texttt{noncricital actions} \\ \quad \texttt{b}_2 \leftarrow \texttt{true}; \\ \quad \texttt{while k} \neq 2 \texttt{ do} \\ \quad\quad \texttt{while b}_1 \texttt{ do} \\ \quad\quad\quad \texttt{skip}; \\ \quad\quad \texttt{k} \leftarrow 2 \\ \quad \texttt{critical actions} \\ \quad \texttt{b}_2 \leftarrow \texttt{false} \end{array}\right]$$

1.1. [2 pt] Implement this algorithm in CCS using two processes in parallel. Use pseuCo, and explain your implementation.

1.2. [2 pt] Implement this algorithm Scala using a thread for each process. **Discuss** your implementation, describing your design decisions and, if applicable, possible alternatives.

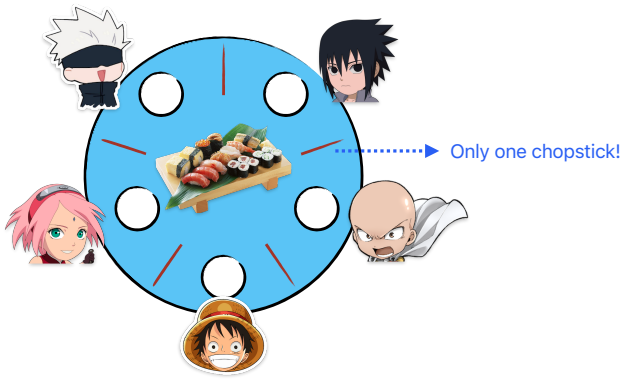
1.3. [1 pt] Test your implementation algorithm using concrete dummy tasks. **Explain** your testing approach, how to run and replicate your experiments, and why we can conclude (or not) that both processes are executing the critical section one at a time.

1.4. [2 pt] Implement an alternative to Hyman's algorithm using Java locks to control the entry to the critical section. **Test** and **explain** your new implementation, using the same dummy tasks as in Exercise 1.3.

1.5. [2 pt] Implement an alternative to Hyman's algorithm using a lock-free approach with an atomic variable to control the entry to the critical section. **Test** and **explain** your new implementation, using the same dummy tasks as in Exercise 1.3.

Sushi meal

Exercise 2. A set of 5 friends go for a sushi meal together and sit in a round table, as in the figure below. Between each friend there is a single chopstick, shared by two friends. During the meal each friend is either eating with his/her left and right chopsticks, or chatting without using his/hers chopsticks, constantly alternating between these two states.



2.1. [2 pt] Encode 3 variations of the sushi-eaters problem as CCS formulas with 5 processes in parallel, one for each friend, and implement them in pseuCo.

- **V1:** Each friend tries to grab the left chopstick (if available), then the right chopstick (if available). After some eating she puts down the chopsticks until she is again hungry, and repeats the same actions.
- **V2:** The same as before, but one of the friends is left-handed, and tries to grab the right chopstick before grabbing the left chopstick.
- **V3:** Before eating, each friend must serve himself a few sushi pieces. More specifically, she must (1) grab the **sushi tray**, (2) grab the **left chopstick**, (3) grab the **right chopstick**, (4) return the **sushi tray**, (5) **eat**, (6) and finally return **both chopsticks**. This repeats forever.

2.2. [1.5 pt] For each of the 3 variations above, **explain** if it can deadlock, and **provide a trace** when applicable.

2.3. [2 pt] Implement V3 using concurrent threads. **Test** your implementation, and explain it (and how to execute it).

2.4. [2.5 pt] Implement an actor system in Scala that encode the sushi eaters from V1 or V2. Use graceful shutdown after a fixed period of time. **Draw** the transition system of the actor classes that you used, **test** your system, and **explain** your implementation (briefly) and how to execute it.

Presentation

Exercise 3. [3 pt] Give a short 8-minute presentation summarising your CCS encodings and your Scala implementations. You do not need to submit slides and all members of the group should talk when giving the presentation. The presentation will be scheduled for June 3 - June 4, 2024.